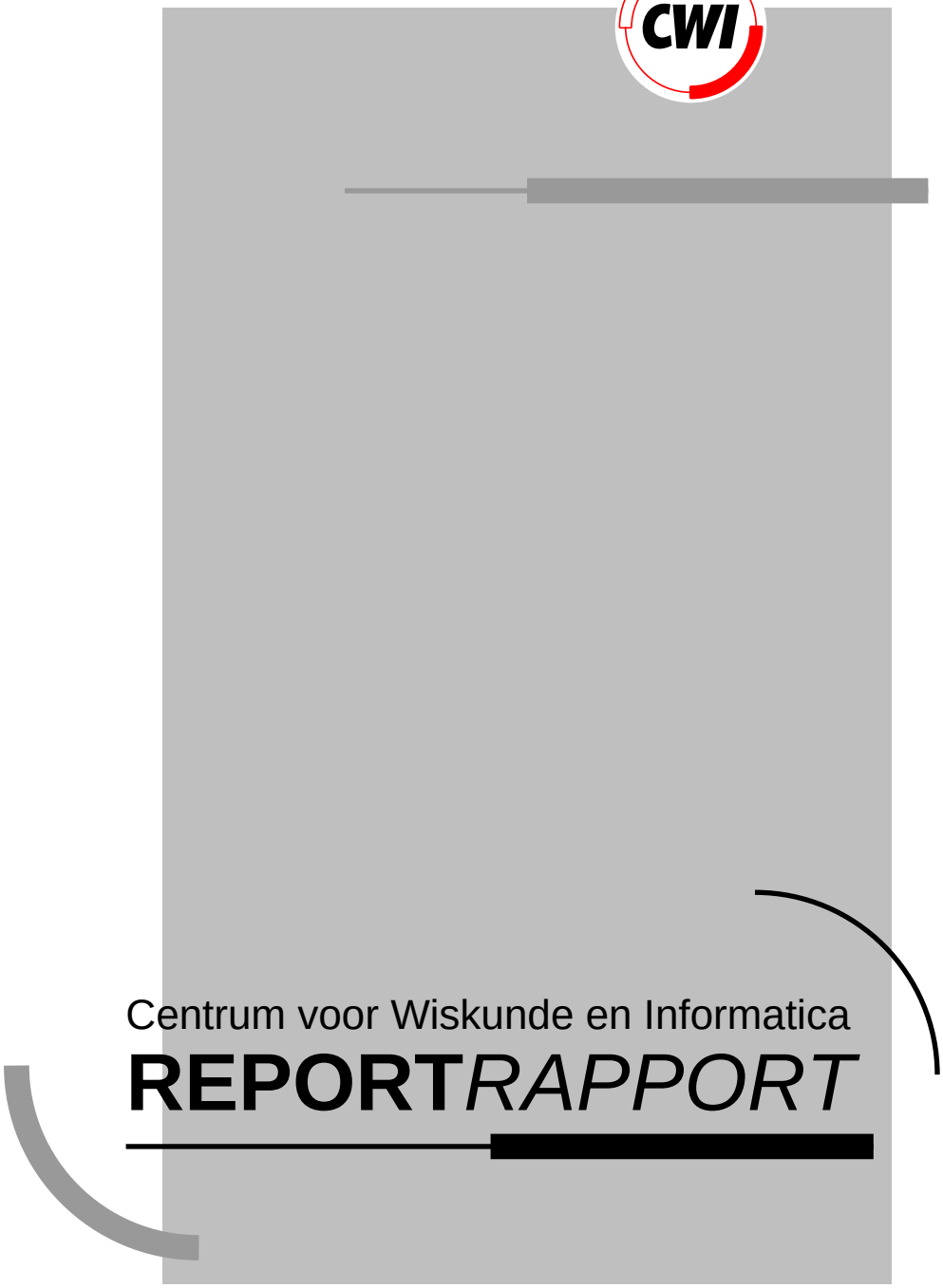


Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

An approach to schema integration based on transformations and behaviour

C Thieme, A Siebes

Computer Science/Department of Algorithmics and Architecture

CS-R9403 1994

An Approach to Schema Integration Based on Transformations and Behaviour

Christiaan Thieme and Arno Siebes
{ct,arno}@cwi.nl

CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

This report presents an approach to schema integration that combines structural aspects and behavioural aspects. The novelty of the approach is that it uses behavioural information to guide both schema restructuring and schema merging. Schema restructuring is based on schema transformations and schema merging is based on join operators.

1991 CR Categories: D.1.5: [Software] Object-oriented programming, D.2.2: [Software] Tools and techniques, H.2.1: [Database management] Logical design.
Keywords and Phrases: Database design, Object-oriented databases, Schema integration.
Note: This research is partly funded by the Dutch Organisation for Scientific Research through NFI-grant NF74.

Contents

1 Introduction

Schema integration is an important and non-trivial task in database design. It occurs when a number of different user views, developed for a new database system, or a number of existing database schemas have to be integrated into a global, unified schema. As schema integration is a difficult task, methods to support the designer with this task are essential. In [7], a framework for comparing integration methods is given. The framework identifies four steps. In the first step, the preintegration step, an integration strategy is chosen and additional information on the schemas is gathered. Subsequently, the schemas are analysed and compared to find similarities/conflicts among the schemas. In the conforming step, the conflicts found in the comparison step have to be resolved. Finally, in the last step, the schemas are merged by superimposition and the resulting schema is analysed and restructured if necessary.

For our purpose, the main characteristic of an integration method is: which similarities/conflicts are detected and how are conflicts resolved? A number of integration methods use assertions among different component schemas to compare attributes and entity types. In [20], interschema assertions, names, and types are used to compare object types. In [18], schemas are merged using schema operators and assertions among entity types and attributes in different schemas. And in [16], attribute assertions (e.g., key/non-key and lower/upper bounds) are used to compare attributes and entity types. However, the assertions must be supplied by the designer and the resolution of conflicts strongly depends on the common sense of the designer.

Other methods use schema transformations to resolve structural conflicts. In [11], structural transformations are defined to integrate compatible structures. In [19], a number of schema transformations (e.g., join and meet) are proposed to restructure schemas. And in [6], transformations between attributes, entities and relationships are used to resolve type conflicts. However, only the last one gives a heuristic (viz., concept likeness/unlikeness) for applying the transformations.

A number of recent methods use more specific information on semantical properties of attributes and entity types to detect similarities and conflicts. In [21, 26], attribute assertions are used to define relationships between an attribute on one hand and a semantic point or a set of concepts on the other hand. Again, the assertions must be supplied by the designer. In [22], a database metadictionary is used to define a semantic domain for each attribute. And in [12], a terminological knowledge base containing information on negative and positive associations between terms and information on specialisation of terms is used to compare entity types.

This report presents a new approach to schema integration, based on schema transformations and the approach taken in [23, 24], where classes are compared by structure and by behaviour. The approach consist of two steps. First, component schemas are restructured using schema transformations, and syntactical properties of methods are used to guide the restructuring process. Subsequently, the component schemas are merged using join operators, and semantical properties of methods are used to guide the merging process. There is, as far as the authors know, no other approach that uses methods to compare attributes. For sake of completeness, it should be mentioned that there is an approach to schema evolution that analyses methods ([9]), not to compare attributes, but to solve non-legitimate overriding of methods.

The outline of this report is as follows. In the next section, database schemas are introduced and formalised in terms of (recursive) types, predicates, and functions. In Section 3, a number of well-known type transformations are extended to recursive types and it is shown how these type transformations induce schema transformations. In Section 4, it is shown how methods can be used to guide schema restructuring and a heuristic algorithm is given to restructure and merge schema. In the last section, a summary and directions for further research are given.

2 Database schemas

In this section, we introduce a subset of the database schemas found in object-oriented database languages such as Galileo [2], Goblin [14], O_2 [17], and TM/FM [4]. Furthermore, we formalise these database schemas in terms of underlying types, underlying constraints, and functional forms.

Informally, an object-oriented database schema is a class hierarchy, i.e., a set of classes related by a subclass relation. A class has a name, a set of superclasses, a set of attributes, a set of constraints, and a set of methods. An attribute has a name and a type, which can be a basic, set, or record type, or a class name. Hence, classes can be recursive. An update method has a name, a list of parameters, and a body, which consists of simple assignments. A query method has a name, a list of input parameters, a result parameter, and a body, which consists of simple assignments.

Definition 1 (Class hierarchies). First, five disjoint sets are postulated: a set CN of class names, a set AN of attribute names, a set MN of method names, a set L of labels, and a set *Cons* of basic constants (i.e., 'integer', 'rational', and 'string' constants). The sets are generated by the nonterminals CN, AN, MN, L, and Cons, respectively. Class hierarchies are the sentences of the following BNF-grammar, where the plus sign ($^+$) denotes a finite, nonempty, repetition, square brackets ([ ]) denote an option, and the vertical bar (|) denotes a choice:

Hierarchy	::=	Class$^+$
Class	::=	'**Class**' CN ['**Isa**' CN$^+$] ['**Attributes**' Att$^+$] ['**Constraints**' Key$^+$] ['**Methods**' Meth$^+$] '**Endclass**'
Att	::=	AN ':' Type
Type	::=	BasicType | SetType | RecordType | CN
BasicType	::=	'integer' | 'rational' | 'string'
SetType	::=	'{' Type '}'
RecordType	::=	'<' FieldList '>'
FieldList	::=	Field | Field ',' FieldList
Field	::=	L ':' Type
Key	::=	'**key**' Dest$^+$
Dest	::=	AN | Dest '.' L
Meth	::=	UpMeth | QueMethod
UpMeth	::=	MN '(' [ParList] ') =' UpAsnList | MN '\' MN '(' [ParList] ') =' UpAsnList
ParList	::=	Par | Par ',' ParList
Par	::=	L ':' BasicType
UpAsnList	::=	UpAssign | UpAssign ';' UpAsnList
UpAssign	::=	UpDest ':=' UpSource | '**insert**(' UpSource ',' UpDest ')'
UpDest	::=	Dest
UpSource	::=	Source | Object '.' MN '(' [ActParList] ')'
Source	::=	'**self**' | Term | Term '+' Source | Term '−' Source | Term '×' Source | Term '÷' Source
Term	::=	Var | Cons
Var	::=	L | AN | Var '.' L | Var '.' AN
Object	::=	'**self**' | Var
ActParList	::=	Source | Source ',' ActParList
QueMeth	::=	MN '(' [ParList] '→' Result ') =' QueAsnList | MN '\' MN '(' [ParList] '→' Result ') =' QueAsnList
Result	::=	L ':' Type
QueAsnList	::=	QueAssign | QueAssign ';' QueAsnList
QueAssign	::=	QueDest ':=' QueSource | '**insert**(' QueSource ',' QueDest ')'
QueDest	::=	L | L '.' QueDest
QueSource	::=	Source | '**new**(' CN ',' NewParList ')'
NewParList	::=	NewPar | NewPar ',' NewParList
NewPar	::=	AN '=' UpSource | AN '= **nself**'

□

A class hierarchy is well-defined if it satisfies four conditions. The first condition is that the **Isa** relation is acyclic (see subsection on underlying types), and classes have a unique name and only refer to classes in the class hierarchy. The second is that attributes have a unique name within their class and are well-typed (see subsection on underlying types). The third is that keys are well-defined (see subsection on underlying constraints). The fourth is that methods have a unique name within their class and are well-typed (see subsection on functional forms).

2.1 Underlying types

In this subsection, we define underlying types of classes, extensions of underlying types, a subtype relation on underlying types, and attribute specialisation.

Example 1. The following class hierarchy introduces a class 'SimpleAddress' and a class 'Address', which inherits from class 'SimpleAddress' and adds a new attribute:

Class SimpleAddress
Attributes
house : integer
street : string
city : string
Endclass
Class Address **Isa** SimpleAddress
Attributes
country : string
Endclass.

□

In definitions, every class is abbreviated to a 5-tuple $C = (c, S, A, K, M)$, where $c = name(C)$ is its name, $S = sup\_names(C)$ is the set of the names of its superclasses, A is the set of its new attributes, K is the set of its new keys, and M is the set of its new methods. Furthermore, every class hierarchy is abbreviated to a set that contains the abbreviations of the classes in the hierarchy.

Informally, the set of all attributes of a class consists of both the new and inherited attributes. In order to formalise this, first, we formalise the **Isa** relation between classes.

Definition 2 (Subclass relation). Let H be an abbreviated class hierarchy satisfying the first condition for well-defined class hierarchies. The subclass relation on H, denoted by $sub(H)$, is defined as the reflexive and transitive closure of:

$$isa(H) = \{(name(C_1), name(C_2)) \mid C_1 \in H \wedge C_2 \in H \wedge name(C_2) \in sup\_names(C_1)\}.$$

Relation $isa(H)$ is acyclic if its transitive closure only contains pairs of the form (c_1, c_2), such that $c_1 \neq c_2$. □

In general, object-oriented data models allow multiple inheritance of attributes, which means that attributes can be inherited from several subclasses. The mechanism defining how attributes are inherited imposes restrictions on multiple inheritance. If the inheritance mechanism is simple, then the restriction on multiple inheritance is severe.

Example 2. Let H be an abbreviated class hierarchy and $C = (c, S, A, K, M)$ be a class in H. Let us define the set of all attributes of C as

$$attributes(C) = A \cup \{a : T \mid \exists C' \in H\ [(name(C), name(C')) \in isa(H) \wedge a : T \in attributes(C')] \wedge\ \forall a' : T' \in A[a \neq a']\}.$$

According to this simple inheritance mechanism, class 'TA' in the following class hierarchy has two different attributes with name 'address':

Class Student
Attributes
name : string
address : <house:integer,street:string,city:string>
Endclass
Class Employee
Attributes
name : string
address : <house:integer,zip:integer>
Endclass
Class TA **Isa** Student Employee
Endclass.

Since we require that attribute names must be unique, the class hierarchy is not well-defined. □

If we use this simple inheritance mechanism, then a necessary condition for class hierarchies to be well-defined is that the types of inherited attributes with the same name must be the same. If we want to relax this restriction, then we must use a more complex inheritance mechanism.

Definition 3 (Attributes). Let H be an acyclic class hierarchy, such that classes have a unique name and only refer to classes in the class hierarchy, and $C = (c, S, A, K, M)$ be a class in H. The set of all attributes of C, denoted by $atts(C)$, is defined as:

$$atts(C) = A \cup \{a : T \mid inherits(a) \wedge T = \sqcap\{T' \mid inherits(a, T')\} \wedge \forall a' : T' \in A[a \neq a']\}$$

where

$$\begin{aligned} inherits(a, T') &= \exists C' \in H\ [(name(C), name(C')) \in isa(H) \wedge a : T' \in atts(C')], \\ inherits(a) &= \exists C' \in H\ \exists a' : T' \in atts(C')\ [(name(C), name(C')) \in isa(H) \wedge a = a'], \end{aligned}$$

and meet operator $\sqcap$ is defined in Appendix A. Since we require that the **Isa** relation is acyclic, $atts$ is well-defined. An attribute $a : T \in atts(C)$ is well-typed if $T \in WTypes$, where $WTypes$ is the smallest set, such that:

1. if $c \in CN$ is a class name, then $c \in WTypes$
2. if $B \in \{\text{integer}, \text{rational}, \text{string}\}$ is a basic type, then $B \in WTypes$
3. if $U \in WTypes$ is a well-defined type, then $\{U\} \in WTypes$
4. if $\{U_1, \cdots, U_n\} \subseteq WTypes$ is a set of well-defined types and $\{l_1, \cdots, l_n\} \subseteq L$ is a set of n distinct labels, then $< l_1 : U_1 \cdots, l_n : U_n > \in WTypes$.

□

Note that $\bot \notin WTypes$. It follows that, in the case of single inheritance, an attribute that is well-typed in class C is also well-typed in every subclass of C. Since we will use the inheritance mechanism of Definition 3 and we do not allow general redefinition of attributes, a necessary condition for class hierarchies to be well-defined is that inherited attributes with the same name must have a meet.

Example 3. Let H be the class hierarchy of Example 2. Let C be class 'TA'. According to the inheritance mechanism of Definition 3. the attributes of class 'Student', of class 'Employee', and of class 'TA', viz.,

$atts(C) = \{$name:string, address:<house:integer, street:string, city:string, zip:integer>$\}$,

are well-typed. Hence, H is well-defined. □

Every class in a class hierarchy has an underlying type, i.e., a type that describes the structure of the objects in its extensions (cf. TM/FM [4, 5]). Since a class can refer to other classes, its underlying type depends on the class hierarchy as a whole. The underlying type of a class is an aggregation of its attributes, where recursive types [3] are used to cope with attributes that refer to classes.

Definition 4 (Underlying types). First, we postulate a new type 'oid', whose extension is an enumerable set of object identifiers. Let H be an acyclic class hierarchy, such that classes have a unique name and only refer to classes in the class hierarchy, and attributes have a unique name within their class and are well-typed. Furthermore, let C be a class in H and c be the name of C. The underlying type of class C, denoted by $type(C)$, is defined as:

$$type(C) = \tau(c, \emptyset)$$

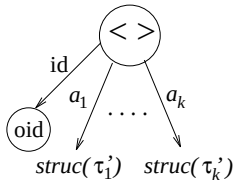
where

$$\begin{array}{l}
\tau(d,\eta) = \mu\, t_d\, .\, < \mathrm{id} : \mathrm{oid}, a_1 : \tau(T_1, \eta \cup \{d\}), \cdots, a_k : \tau(T_k, \eta \cup \{d\}) > \\
\qquad \text{if } d \notin \eta \text{ and } \exists D \in H[name(D) = d \wedge atts(D) = \{a_1 : T_1, \cdots, a_k : T_k\}], \\
\tau(d,\eta) = t_d \ \text{ if } d \in \eta, \\
\tau(B,\eta) = B \ \text{ if } B \in \{\mathrm{integer}, \mathrm{rational}, \mathrm{string}\}, \\
\tau(\{U\},\eta) = \{\tau(U,\eta)\}, \\
\tau(< l_1 : U_1, \cdots, l_n : U_n >, \eta) = < l_1 : \tau(U_1,\eta), \cdots, l_n : \tau(U_n,\eta) >.
\end{array}$$

The set η contains the names of the classes for which a (recursive) type is being constructed as part of the construction of the underlying type of class C. If η contains d, then $\tau(d,\eta) = t_d$ indicates a repetition of the recursive type. □

A natural equivalence relation for types is the following [15]: τ is equivalent to τ' if the tree representing τ is equal to the tree representing τ'.

Definition 5 (Type equivalence). Let τ be type $\mu t.\ < \mathrm{id} : \mathrm{oid}, a_1 : \tau_1, \cdots, a_k : \tau_k >$ and τ_i' be $\tau_i[t \setminus \tau]$ for $i \in \{1, \cdots, k\}$. The tree representing τ, denoted by $struc(\tau)$, is defined as:



where

- $struc(B)$ has only one node, labeled B, if $B \in \{\mathrm{integer}, \mathrm{rational}, \mathrm{string}\}$
- $struc(\{v\})$ consists of a root, labeled $\{\}$, a subtree $struc(v)$, and an arrow, labeled $\in$, from the root labeled $\{\}$ to the root of $struc(v)$,
- $struc(< l_1 : v_1, \cdots, l_n : v_n >)$ consists of a root, labeled $<>$, subtrees $struc(v_1), \cdots, struc(v_n)$, and arrows, labeled l_i, one for each $i \in \{1, \cdots, n\}$, from the root labeled $<>$ to the root of $struc(v_i)$.

Now, let τ and τ' be arbitrary types. Equivalence of τ and τ' is defined as equality of their trees:

$$\tau \cong \tau' \ \Leftrightarrow \ struc(\tau) = struc(\tau').$$

□

Example 4. The underlying types of class 'SimpleAddress' and class 'Address' of Example 1 are given by:

$\tau_S = \mu\, t_S$. <id:oid, house:integer, street:string, city:string>
$\cong$ <id:oid, house:integer, street:string, city:string>,
$\tau_A = \mu\, t_A$. <id:oid, house:integer, street:string, city:string, country:string>
$\cong$ <id:oid, house:integer, street:string, city:string, country:string>.

Using the definition and the rule $\mu t.\alpha \cong \alpha[t \setminus \mu t.\alpha]$, where $\alpha[t \setminus e]$ means that every occurrence of t in α must be replaced by e, we have obtained types that are equivalent to the underlying types in TM/FM. □

Example 5. The following class hierarchy introduces a class 'Person' and a class 'Employee', which inherits from class 'Person':

Class Person
Attributes
name : string
mother : Person
address : SimpleAddress
holiday_address : SimpleAddress
Endclass
Class Employee **Isa** Person
Attributes
company : string
salary : integer
Endclass.

Let α be <id:oid, name:string, mother:t_P, address:τ_S, holiday_address:τ_S >, where t_P is a type variable and τ_S is the underlying type of class 'Simple_Address'. The underlying types of class 'Person' and class 'Employee' are given by:

$\tau_P = \mu\, t_P$. α
$\cong$ <id:oid, name:string, mother:$\mu\, t_P.\alpha$, address:τ_S, holiday_address:τ_S >,
$\tau_E = \mu\, t_E$. <id:oid, name:string, mother:$\mu\, t_P.\alpha$, address:τ_S, holiday_address:τ_S,
company:string, salary:integer>
$\cong$ <id:oid, name:string, mother:$\mu\, t_P.\alpha$, address:τ_S, holiday_address:τ_S,
company:string, salary:integer>.

Using the definition and the rule $\mu t.\alpha \cong \alpha[t \setminus \mu t.\alpha]$, we have obtained underlying types that are not equivalent to the underlying types in TM/FM, which are <id:oid, name:string, mother:oid, address:oid, holiday_address:oid> and <id:oid, name:string, mother:oid, address:oid, holiday_address: oid, company:string, salary:integer>, respectively. □

The extension of a type is the set of closed terms of that type.

Definition 6 (Extensions). Let τ be an arbitrary type. The set of terms of type τ, denoted by $terms(\tau)$, is defined as follows:

$terms(B) = \{b \in Cons \mid b \text{ has type } B\}$ if $B \in \{\text{oid}, \text{integer}, \text{string}, \text{rational}\}$
$terms(\{v\}) = \{e \subseteq terms(v) \mid e \text{ is finite}\}$
$terms(< l_1 : v_1, \cdots, l_n : v_n >) =$
$\{< l_1 = e_1, \cdots, l_n = e_n > \mid e_1 \in terms(v_1) \wedge \cdots \wedge e_n \in terms(v_n)\}$
$terms(\mu t.\alpha) = Var_t \cup \{\mu x.e \mid x \in (Var_t - BV(e)) \wedge e \in terms(\alpha[t \setminus \mu t.\alpha])\}$

where Var_t is an enumerable set of instance variables, disjoint from Var_s for $s \neq t$, and $BV(e)$ is the set of bounded variables in term e:

$BV(b) = \emptyset$ if $b \in Cons$,
$BV(y) = \emptyset$ if $y \in Var_s$ for some s,
$BV(\{e_1, \cdots, e_n\}) = BV(< l_1 = e_1, \cdots, l_n = e_n >) = BV(e_1) \cup \cdots \cup BV(e_n)$,
$BV(\mu y.e) = BV(e) \cup \{y\}$.

The extension of type τ, denoted by $ext(\tau)$, is defined as:

$$ext(\tau) = \{e \in terms(\tau) \mid FV(e) = \emptyset\}$$

where $FV(e)$ is the set of free variables in term e:

$FV(b) = \emptyset$ if $b \in Cons$,
$FV(y) = \{y\}$ if $y \in Var_s$ for some s,
$FV(\{e_1, \cdots, e_n\}) = FV(< l_1 = e_1, \cdots, l_n = e_n >) = FV(e_1) \cup \cdots \cup FV(e_n)$,
$FV(\mu y.e) = FV(e) - \{y\}$.

□

A natural equality relation for closed terms is the following: e is equal to e' if the tree representing e is equal to the tree representing e'.

Definition 7 (Equality of terms). Let e and e' be closed terms of arbitrary types. Equality of e and e' is defined as equality of their trees:

$$e = e' \Leftrightarrow struc(e) = struc(e'),$$

where

- $struc(b)$ has only one node, labeled b, if $b \in Cons$,
- $struc(\emptyset)$ has only one node, labeled $\{\}$,
- $struc(\{e_1, \cdots, e_n\})$ consists of a root, labeled $\{\}$, subtrees $struc(e_1), \cdots, struc(e_n)$, and arrows, labeled $\in$, one for each $i \in \{1, \cdots, n\}$, from the root labeled $\{\}$ to the root of $struc(e_i)$,
- $struc(< l_1 = e_1, \cdots, l_n = e_n >)$ consists of a root, labeled $<>$, subtrees $struc(e_1), \cdots, struc(e_n)$, and arrows, labeled l_i, one for each $i \in \{1, \cdots, n\}$, from the root labeled $<>$ to the root of $struc(e_i)$,
- $struc(\mu x.e) = struc(e[x \setminus \mu x.e])$.

□

A natural subtype relation for underlying types is the following [3, 8]: $\mu t.\alpha$ is a subtype of $\mu t'.\alpha'$ if from the assumption that t is a subtype of t' it follows that for every field $a : \upsilon'$ in α' there is a field $a : \upsilon$ in α, such that υ is a subtype of υ'.

Definition 8 (Subtyping). Let underlying types τ and τ' be given by:

$\tau = \mu t. < l_i : \tau_i \mid i \in I >$,
$\tau' = \mu t'. < l_i : \tau'_i \mid i \in I' >$.

Then τ is a subtype of τ', denoted by $\tau \leq \tau'$, if

$$I \supseteq I' \wedge \forall i \in I'[t \leq t' \Rightarrow \tau_i \leq \tau'_i],$$

where

$B \leq B$ if $B \in \{\text{oid}, \text{integer}, \text{rational}, \text{string}\}$,
$\{\upsilon\} \leq \{\upsilon'\}$ if $\upsilon \leq \upsilon'$,
$< l_j : \upsilon_j \mid j \in J > \leq < l_j : \upsilon'_j \mid j \in J' >$ if $J \supseteq J' \wedge \forall j \in J'[\upsilon_j \leq \upsilon'_j]$.

□

Example 6. Let τ_S and τ_A be the types given in Example 4 and τ_P and τ_E be the types given in Example 5. Using the subtype rules, it follows that τ_A is a subtype of τ_S and τ_E is equivalent to a subtype of τ_P. □

Attribute specialisation is a form of attribute redefinition, where an inherited attribute $a : T$, defined in class C, is replaced by $a : T'$ in subclass C', such that $type(T')$ is a subtype of $type(T)$. If attribute specialisation is allowed, it still holds that the underlying type of a subclass is a subtype of the underlying type of the superclass (in case of general redefinition of attributes, it does not hold). Now we can reformulate the second condition for well-defined class hierarchies: every attribute must have a unique name within its class, every attribute must be well-typed, and the type of every inherited attribute must be a subtype of the types of the corresponding attributes in the superclasses.

2.2 Underlying constraints

In this subsection, we define underlying constraints of classes and class extensions.

Example 7. The following class hierarchy introduces a class 'Person' with a constraint and a class 'Employee', which inherits from class 'Person':

Class Person
Attributes
name : string
mother : Person
address : SimpleAddress
holiday_address : SimpleAddress
Constraints
key name
Endclass
Class Employee **Isa** Person
Attributes
company : string
salary : integer
Endclass.

□

Informally, the set of all keys of a class consists of both the new and inherited keys.

Definition 9 (Keys). Let H be an acyclic class hierarchy, such that classes have a unique name and only refer to classes in the class hierarchy, and $C = (c, S, A, K, M)$ be a class in H. The set of all keys of C is defined as:

$$keys(C) = K \cup \{p \mid \exists C' \in H\ [(name(C), name(C')) \in isa(H) \wedge p \in keys(C')]\}.$$

A key p in $keys(C)$ is well-defined if it is a sequence of labeling of paths in $struc(type(C))$, starting at the root. □

It follows that a key that is well-defined in class C is also well-defined in every subclass of C.

Every class in a class hierarchy has an underlying constraint, i.e., a predicate that is satisfied by its extensions. The underlying constraint of a class is a conjunction of identifier uniquess, partial referential integrity, and key uniqueness.

Definition 10 (Underlying constraints). Let H be an acyclic class hierarchy, such that classes have a unique name and only refer to classes in the class hierarchy, and C be a class in H. The set of extensions of class C is defined as:

$$exts(C) = \{e \subseteq ext(type(C)) \mid constraint(C, e)\},$$

where $constraint(C, e)$ is defined as the conjunction of

1. identifier uniqueness: $\forall x \in e\ \forall y \in e\ [x.id = y.id \Rightarrow x = y]$

2. partial referential integrity: one formula for every occurrence of $name(C)$ in $atts(C)$

3. key uniqueness: one formula for every key $p_1 \cdots p_n$ in $keys(C)$:
   $\forall x \in e\ \forall y \in e\ [(x.p_1 = y.p_1 \wedge \cdots \wedge x.p_n = y.p_n) \Rightarrow x = y]$.

The formulas for referential integrity are defined in Appendix B. □

2.3 Functional forms

In this subsection, we define functional forms of methods.

Example 8. The following class hierarchy introduces a class 'Person' with an update method 'stay' and a class 'Employee', which inherits from class 'Person':

Class Person
Attributes
  name : string
  mother : Person
  address : SimpleAddress
  holiday_address : SimpleAddress
Methods
  stay() = holiday_address := address
Endclass
Class Employee **Isa** Person
Attributes
  company : string
  salary : integer
Endclass.

□

Informally, the set of all methods of a class consists of both the new and inherited methods.

Definition 11 (Methods). Let H be an acyclic class hierarchy, such that classes have a unique name and only refer to classes in the class hierarchy, and attributes have a unique name within their class and are well-typed, and $C = (c, S, A, K, M)$ be a class in H. The set of all methods of C, denoted by $meths(C)$, is defined as:

$$meths(C) = M \cup \{m(P) = E \mid \exists C' \in H\ [(name(C), name(C')) \in isa(H) \wedge m(P) = E \in meths(C')] \wedge \forall m'(P') = E' \in M[m \neq m']\}.$$

Let $m(P \rightarrow l : T) = E$ be a query method in $meths(C)$. The query method is well-typed if for every assignment $d := s$ in E:

1. d starts with l (i.e., only assignments to the result of the method)

2. if s is $\mathbf{new}(c', a_1 = e_1, \cdots, a_n = e_n)$, such that c' is the name of $C' \in H$, then:

   (a) $(name(C), c')$ is an element of $sub(H)$
   (i.e., only creation of objects in class C and superclasses of C)

   (b) $\mid atts(C') \mid = n$

   (c) for every $i \in \{1, \cdots, n\}$, $a_i : T_i$ is an attribute in $atts(C')$ and the type of e_i is a subtype of T_i according to subtype relation $\leq_H$

3. the type of s is a subtype of the type of d according to subtype relation $\leq_H$
   (i.e., only assignments of objects that belong to the corresponding class and its subclasses)

where the types of sources and destinations are defined in Appendix C, and subtype relation $\leq_H$ is an extension of subclass relation $sub(H)$:

$d \leq_H d'$ if $(d, d') \in sub(H)$
$B \leq_H B$ if $B \in \{\text{oid}, \text{integer}, \text{rational}, \text{string}\}$,
$\{U\} \leq_H \{U'\}$ if $U \leq_H U'$,
$< l_j : U_j \mid j \in J > \leq_H < l_j : U'_j \mid j \in J' >$ if $J \supseteq J' \wedge \forall j \in J'[U_j \leq_H U'_j]$.

Now, let $m(P) = E$ be an update method in $meths(C)$. The update method is well-typed if for every assignment $d := s$ in E, the type of s is a subtype of the type of d (according to $\leq_H$) and if for every method call $s.m'(v_1, \cdots, v_n)$ in E, the following holds: the type of s must be $name(D)$ for some $D \in H$ and there must be a method $m'(P' \rightarrow l' : T') = E'$ in $meths(D)$, such that the type of each v_i is a subtype (according to $\leq_H$) of the type of the corresponding formal parameter in P'. □

Since we will use the inheritance mechanism of Definition 11 for methods, a necessary condition for class hierarchies to be well-defined is that inherited methods with the same name must be the same or must be redefined.

Example 9. According to the inheritance mechanism of Definition 11, class 'Employee' in the following class hierarchy has one method, viz., 'stay() = holiday_address := address':

Class Person
Attributes
name : string
mother : Person
address : SimpleAddress
holiday_address : SimpleAddress
Methods
stay() = holiday_address := address
Endclass
Class Employee **Isa** Person
Attributes
holiday_address : Address
company : string
salary : integer
Endclass.

Although method 'stay' is well-typed in class 'Person', it is not well-typed in class 'Employee', because attribute 'holiday_address' has been specialised (if we do not allow attribute specialisation, then every method that is well-typed in class C is also well-typed in every subclass of C). Hence, class 'Employee' is not well-defined. One way to repair this is to use a different inheritance mechanism for methods, where ill-typed methods are redefined. However, we do not allow general redefinition of methods. Nevertheless, class 'Employee' can be redefined as a well-defined class by specialising attribute 'address' in the same way as attribute 'holiday_address', i.e., adding attribute 'address:Address' to the new attributes of class 'Employee'. For a more complicated example, let class 'Person' and class 'AgedPerson' be given by:

Class Person
Attributes
name : string
address : SimpleAddress
mother : Person
grandmother : Person
Methods
define_grandmother () =
grandmother := mother.mother1()
mother1 (→ p:Person) =
p := mother

Endclass
Class AgedPerson **Isa** Person
Attributes
address : Address
grandmother : AgedPerson
Endclass.

Method 'define_grandmother' is well-typed in class 'Person', but not in class 'AgedPerson', because attribute 'grandmother' has been specialised. Hence, class 'AgedPerson' is not well-defined. However, class 'AgedPerson' can be redefined as a well-defined class by specialising both attribute 'mother' and method 'mother1':

Class AgedPerson **Isa** Person
Attributes
address : Address
mother : AgedPerson
grandmother : AgedPerson
Methods
define_grandmother () =
grandmother := mother.mother1()
mother1 ($\rightarrow$ p:AgedPerson) =
p := mother
Endclass.

□

Every class in a class hierarchy has a set of functional forms (one for each of its methods), i.e., a set of functions that describe the way in which the objects in its extensions are queried and updated (cf. TM/FM [4, 10]). The functional form of a query (resp., an update) method is a function of which the body is an accumulation of the assignments to the result (resp., to the input) of the method.

Definition 12 (Functional forms). Let $H = \{C_1, \cdots, C_h\}$ be an acyclic class hierarchy, such that classes have a unique name and only refer to classes in the class hierarchy, and attributes have a unique name within their class and are well-typed. Let $CN(H) = \{c_1, \cdots, c_h\}$ be the set of class names in H, C be a class in H, c be $name(C)$, and $\{a_1 : T_1, \cdots, a_k : T_k\}$ be $atts(C)$. Furthermore, let $meth = m(P \rightarrow l : T) = E$ be a query method in $meths(C)$. The functional form of query method $meth$ in class C, denoted by $func_q(C, meth)$, is defined as:

$$func_q(C, meth) =$$
$$\lambda objcount : (CN(H) \rightarrow \text{integer})\ \lambda(obj) : type(C)\ \lambda P\ .$$
$$eval_q(E)(obj, \rho_0(T), objcount),$$

where

$\rho_0(v, d) = \ \perp$ if $d \in CN$,
$\rho_0(v, B) = \ \perp$ if $B \in \{\text{integer}, \text{rational}, \text{string}\}$,
$\rho_0(v, \{U\}) = \ \perp$,
$\rho_0(v, < l_1 : U_1, \cdots, l_n : U_n >) = < l_1 = \rho_0(v.l_1, U_1), \cdots, l_n = \rho_0(v.l_n, U_n) >$,

and syntactic evaluation function $eval_q$ is defined in Appendix D. Let $\lambda p_1 : \tau_1 \cdots \lambda p_n : \tau_n$ be λP. The retrieval semantics of $meth$ are given by the function:

$$\lambda db : < c_1 : \{type(C_1)\}, \cdots, c_h : \{type(C_h))\} >\ \lambda obj : type(C)\ \lambda p_1 : \tau_1 \cdots \lambda p_n : \tau_n.$$
$$\pi_2(F(db, obj)),$$

where $F(db, obj)$ is $(func_q(C, meth))(\lambda x : CN(H).\ |\ db.x\ |)(obj)(p_1) \cdots (p_n)$ and π_i is the projection of the ith component of a record.

Now, let $meth = m(P) = E$ be an update method in $meths(C)$. The functional form of update method $meth$ in class C, denoted by $func_u(C, meth)$, is defined as:

$$
\begin{array}{l}
func_u(C, meth) = \\
\quad \lambda objcount : (CN(H) \to \text{integer})\ \lambda(\mu x.e) : type(C)\ \lambda P\ . \\
\qquad eval_u(E)(\mu x. < \text{id} = e.\text{id}, a_1 = \sigma_0(a_1, T_1), \cdots, a_k = \sigma_0(a_k, T_k) >, objcount),
\end{array}
$$

where

$$
\begin{array}{l}
\sigma_0(v, d) = e.v \ \text{ if } d \in CN, \\
\sigma_0(v, B) = e.v \ \text{ if } B \in \{\text{integer}, \text{rational}, \text{string}\}, \\
\sigma_0(v, \{U\}) = e.v, \\
\sigma_0(v, < l_1 : U_1, \cdots, l_n : U_n >) = < l_1 = \sigma_0(v.l_1, U_1), \cdots, l_n = \sigma_0(v.l_n, U_n) >,
\end{array}
$$

and syntactic evaluation function $eval_u$ is defined in Appendix D. Let $\lambda p_1 : \tau_1 \cdots \lambda p_n : \tau_n$ be λP. The local update semantics of $meth$ are given by the function:

$$
\begin{array}{l}
\lambda db : < c_1 : \{type(C_1)\}, \cdots, c_h : \{type(C_h))\} >\ \ \lambda obj : type(C)\ \lambda p_1 : \tau_1 \cdots \lambda p_n : \tau_n. \\
\qquad\qquad \pi_1(F(db, obj))
\end{array}
$$

where $F(db, obj)$ is $(func_u(C, meth))(\lambda x : CN(H).\ |\ db.x\ |)(obj)(p_1) \cdots (p_n)$. □

Example 10. Let H be the class hierarchy of Example 9. Let C_P be class 'Person' and C_E be class 'Employee'. The local update semantics of method 'stay' in class 'Person' and class 'Employee' are given by:

$$
\begin{array}{l}
\lambda db : < \text{Person} : \{type(C_P)\}, \text{Employee} : \{type(C_E)\} > \lambda(\mu x.e) : type(C_P)\ . \\
\quad \mu x.\ <\text{id}=e.\text{id}, \text{name}=e.\text{name}, \text{mother}=e.\text{mother}, \\
\qquad\qquad \text{address}=e.\text{address}, \text{holiday\_address}=e.\text{address}>, \\
\lambda db : < \text{Person} : \{type(C_P)\}, \text{Employee} : \{type(C_E)\} > \lambda(\mu x.e) : type(C_E)\ . \\
\quad \mu x.\ <\text{id}=e.\text{id}, \text{name}=e.\text{name}, \text{mother}=e.\text{mother}, \text{address}=e.\text{address}, \\
\qquad\qquad \text{holiday\_address}=e.\text{address}, \text{company}=e.\text{company}, \text{salary}=e.\text{salary}>.
\end{array}
$$

□

3 Schema transformations

In this section, we give an overview of type transformations and show how type transformations induce schema transformations.

The basic type transformations we have chosen (viz., renaming, aggregation, and objectification) are variants of type transformations in [1].

Definition 13 (Basic type transformations). Let L' be the union of L and AN and $Types$ be the set of types. Renaming is defined as a function of type $L' \to L' \to Types \to Types$:

$$
\begin{array}{l}
rename(l')(l)(B) = B \text{ if } B \in \{\text{oid}, \text{integer}, \text{rational}, \text{string}\} \\
rename(l')(l)(\{\tau\}) = \{\tau\} \\
rename(l')(l)(< l_1 : \tau_1, \cdots, l_n : \tau_n >) = < l_1[l' \setminus l] : \tau_1, \cdots, l_n[l' \setminus l] : \tau_n >, \\
rename(l')(l)(\mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >) = \mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n > \\
\quad \text{if } l' = \text{id}, \\
rename(l')(l)(\mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >) = \mu t. < l_1[l' \setminus l] : \tau_1, \cdots, l_n[l' \setminus l] : \tau_n > \\
\quad \text{if } l' \neq \text{id}.
\end{array}
$$

Note that we do not allow renaming of id-fields.
We distinguish between two kinds of aggregation: simple tupling and aggregation within a record type. Simple tupling is defined as a function of type $L' \to Types \to Types$:

$$
tuple(l)(\tau) = < l : \tau >.
$$

Aggregation within a record type is defined as a function of type $\{L'\} \to L' \to Types \to Types$:

$aggregate(\{l_i, l_{i+1}, \cdots, l_j\})(l)(B) = B$ if $B \in \{\text{oid}, \text{integer}, \text{rational}, \text{string}\}$
$aggregate(\{l_i, l_{i+1}, \cdots, l_j\})(l)(\{\tau\}) = \{\tau\}$
$aggregate(\{l_i, l_{i+1}, \cdots, l_j\})(l)(< l_1 : \tau_1, \cdots, l_n : \tau_n >) = < l_1 : \tau_1, \cdots, l_n : \tau_n >$
if $\{l_i, l_{i+1}, \cdots, l_j\} \not\subseteq \{l_1, \cdots, l_n\}$,
$aggregate(\{l_i, l_{i+1}, \cdots, l_j\})(l)(< l_1 : \tau_1, \cdots, l_n : \tau_n >) =$
$< l_1 : \tau_1, \cdots, l :< l_i : \tau_i, \cdots, l_j : \tau_j >, \cdots, l_n : \tau_n >$
if $\{l_i, l_{i+1}, \cdots, l_j\} \subseteq \{l_1, \cdots, l_n\}$,
$aggregate(\{l_i, l_{i+1}, \cdots, l_j\})(l)(\mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >) = \mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >$
if $\{l_i, l_{i+1}, \cdots, l_j\} \not\subseteq \{l_1, \cdots, l_n\}$,
$aggregate(\{l_i, l_{i+1}, \cdots, l_j\})(l)(\mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >) =$
$\mu t. < l_1 : \tau_1, \cdots, l :< l_i : \tau_i, \cdots, l_j : \tau_j >, \cdots, l_n : \tau_n >$
if id $\notin \{l_i, l_{i+1}, \cdots, l_j\}$ and $\{l_i, l_{i+1}, \cdots, l_j\} \subseteq \{l_1, \cdots, l_n\}$,
$aggregate(\{l_i, l_{i+1}, \cdots, l_j\})(l)(\mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >) =$
$\mu s. < l_1 : \tau_1[t \setminus s], \cdots, l : \mu t. < l_i : \tau_i[t \setminus s], \cdots, l_j : \tau_j[t \setminus s] >, \cdots, l_n : \tau_n[t \setminus s] >$
if id $\in \{l_i, l_{i+1}, \cdots, l_j\}$ and $\{l_i, l_{i+1}, \cdots, l_j\} \subseteq \{l_1, \cdots, l_n\}$.

Objectification is defined as a function of type $Types \rightarrow Types$:

$objectify(B) = B$ if $B \in \{\text{oid}, \text{integer}, \text{rational}, \text{string}\}$
$objectify(\{\tau\}) = \{\tau\}$
$objectify(< l_1 : \tau_1, \cdots, l_n : \tau_n >) = < \text{id} : \text{oid}, l_1 : \tau_1, \cdots, l_n : \tau_n >,$
$objectify(\mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >) = \mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >$
if $\exists i \in \{1, \cdots, n\}[l_i = \text{id}]$,
$objectify(\mu t. < l_1 : \tau_1, \cdots, l_n : \tau_n >) = \mu t. < \text{id} : \text{oid}, l_1 : \tau_1, \cdots, l_n : \tau_n >$
if $\forall i \in \{1, \cdots, n\}[l_i \neq \text{id}]$.

□

Complex type transformations are obtained by combining basic type transformations.

Example 11. Type $\sigma = < l_1 : \mu t. < \text{id} : \text{oid}, l : \tau_1, l_2 : \tau_2 >, l_3 : \tau_3 >$ can be obtained from type $\sigma_1 = < l_1 : \tau_1, l_2 : \tau_2, l_3 : \tau_3 >$ as follows:

$\sigma_2 = rename(l_1)(l)(\sigma_1) = < l : \tau_1, l_2 : \tau_2, l_3 : \tau_3 >$
$\sigma_3 = aggregate(\{l, l_2\})(l_1)(\sigma_2) = < l_1 : < l : \tau_1, l_2 : \tau_2 >, l_3 : \tau_3 >$
$\sigma_4 = < l_1 : objectify(< l : \tau_1, l_2 : \tau_2 >), l_3 : \tau_3 > = \sigma.$

□

If F is a type transformation and τ is a type, then F induces a function from $struc(\tau)$ to $struc(F(\tau))$.

Definition 14 (Functions between trees). Let F be a rename operation $rename(l_1)(l')$ and τ be a type, such that $F(\tau) \not\equiv F(\tau)$, where $\equiv$ is syntactical equality. Then F induces a function Φ from the set of paths in $struc(\tau)$ to the set of paths in $struc(F(\tau))$. Function Φ is defined by $\Phi(p) = \varphi(p, \tau)$, where

$\varphi(l, \upsilon) = l'$ if $l = l_1$ and $\upsilon \equiv \tau$
$\varphi(l, \upsilon) = l$ if $l \neq l_1$ or $\upsilon \not\equiv \tau$
$\varphi(l.l_1.\cdots.l_n, \upsilon) = \varphi(l, \upsilon).\varphi(l_1.\cdots.l_n, \upsilon.l),$

where

$< \cdots, l : \tau, \cdots >.l = \tau,$
$(\mu t.\alpha).l = (\alpha.l)[t \setminus \mu t.\alpha].$

Let F be a tupling operation $tuple(l')$ and τ be a type, such that $F(\tau) \not\equiv F(\tau)$. Then F induces a function Φ from the set of paths in $struc(\tau)$ to the set of paths in $struc(F(\tau))$. Function Φ is defined by $\Phi(p) = \varphi(p, \tau)$, where

$\varphi(l, \upsilon) = l'.l$ if $\upsilon \equiv \tau$
$\varphi(l, \upsilon) = l$ if $\upsilon \not\equiv \tau$
$\varphi(l.l_1.\cdots.l_n, \upsilon) = \varphi(l, \upsilon).\varphi(l_1.\cdots.l_n, \upsilon.l)$.

Let F be an aggregation operation $aggregate(\{l'_1, \cdots, l'_p\})(l')$ and τ be a type, such that $F(\tau) \not\equiv F(\tau)$. Then F induces a function Φ from the set of paths in $struc(\tau)$ to the set of paths in $struc(F(\tau))$. Function Φ is defined by $\Phi(p) = \varphi(p, \tau)$, where

$\varphi(l, \upsilon) = l'.l$ if $l \in \{l'_1, \cdots, l'_p\}$ and $\upsilon \equiv \tau$
$\varphi(l, \upsilon) = l$ if $l \notin \{l'_1, \cdots, l'_p\}$ or $\upsilon \not\equiv \tau$
$\varphi(l.l_1.\cdots.l_n, \upsilon) = \varphi(l, \upsilon).\varphi(l_1.\cdots.l_n, \upsilon.l)$.

Let F be an objectify operation and τ be a type, such that $F(\tau) \not\equiv F(\tau)$. Then F induces a function Φ from the set of paths in $struc(\tau)$ to the set of paths in $struc(F(\tau))$, which is defined by $\Phi(p) = p$. □

Type transformations induce transformations on predicates.

Definition 15 (Transformations on predicates). Let F be a type transformation, τ be a type, and ψ be a conjunction of formulas, such that each formula has one of the following forms:

1. uniqueness: $\forall x \in e \forall y \in e[(x.p_1 = y.p_1 \wedge \cdots \wedge x.p_n = y.p_n) \Rightarrow x = y]$
2. refint: $\forall x \in e \forall x_1 \in x.p_1 \cdots \forall x_n \in x_{n-1}.p_n \exists y \in e[x_n.p_{n+1} = y]$.

Predicate $F(\psi)$ is obtained from ψ by replacing each formula in ψ as follows:

1. for each uniqueness formula: for every p_i, if p_i is a path in $struc(\tau)$, then p_i is replaced by $\Phi(p_i)$
2. for each refint formula: for every p_i, if $p_1. \in .\cdots. \in .p_i$ is a path in $struc(\tau)$, then p_i is replaced by q_i, such that $\Phi(p_1. \in .\cdots. \in .p_i) = \Phi(p_1. \in .\cdots. \in .p_{i-1}).\epsilon.q_i$,

where Φ is the function from the set of paths in $struc(\tau)$ to the set of paths in $struc(F(\tau))$ induced by F. □

Type transformations also induce transformations on functions.

Definition 16 (Transformations on functions). Let type τ and function f be given by:

$\tau = \mu t. <\text{id} : \text{oid}, a :< l_1 : \tau_1, \cdots, l_n : \tau_n >>$
$f = \lambda(\mu x.e) : \tau\ \lambda P.\ \mu x. <\text{id} = e_0, a = < l_1 = e_1, \cdots, l_n = e_n >>$.

Let F be a rename operation for types, such that:

$F(\tau) = \mu t. <\text{id} : \text{oid}, a : rename(l_1)(l)(< l_1 : \tau_1, \cdots, l_n : \tau_n >) >$.

Then F induces a function $F(f)$ on $F(\tau)$, defined by:

$F(f) = \lambda(\mu x.e) : F(\tau)\ \lambda P.$
$\quad \mu x. <\text{id} = \Phi(e_0), a = < l = \Phi(e_1), l_2 = \Phi(e_2), \cdots, l_n = \Phi(e_n) >>$,

where Φ is the function from $struc(\tau)$ to $struc(F(\tau))$ induced by F. Let F be a tupling operation for types, such that:

$F(\tau) = \mu t. <\text{id} : \text{oid}, a : tuple(l)(< l_1 : \tau_1, \cdots, l_n : \tau_n >) >$.

Then F induces a function $F(f)$ on $F(\tau)$, defined by:

$F(f) = \lambda(\mu x.e) : F(\tau)\ \lambda P.$
$\quad \mu x. <\text{id} = \Phi(e_0), a = < l = < l_1 = \Phi(e_1), \cdots, l_n = \Phi(e_n) >>>$,

where Φ is the function from $struc(\tau)$ to $struc(F(\tau))$ induced by F. Let F be an aggregation operation for types, such that:

$F(\tau) = \mu t. < \text{id} : \text{oid}, a : aggregate(\{l_1, l_2, \cdots, l_p\})(l)(< l_1 : \tau_1, \cdots, l_n : \tau_n >) >.$

Then F induces a function $F(f)$ on $F(\tau)$, defined by:

$F(f) = \lambda(\mu x.e) : F(\tau)\ \lambda P.$
$\quad \mu x. < \text{id} = \Phi(e_0), a = < l = < l_1 = \Phi(e_1), \cdots, l_p = \Phi(e_p) >, \cdots, l_n = \Phi(e_n) >>,$

where Φ is the function from $struc(\tau)$ to $struc(F(\tau))$ induced by F. Let F be an objectify operation for types, such that:

$F(\tau) = \mu t. < \text{id} : \text{oid}, a : objectify(< l_1 : \tau_1, \cdots, l_n : \tau_n >) >.$

Then F induces a function $F(f)$ on $F(\tau)$, defined by:

$F(f) = \lambda(\mu x.e) : F(\tau)\ \lambda P.$
$\quad \mu x. < \text{id} = \Phi(e_0), a = < \text{id} = e.a.\text{id}, l_1 = \Phi(e_1), \cdots, l_n = \Phi(e_n) >>,$

where Φ is the function from $struc(\tau)$ to $struc(F(\tau))$ induced by F. □

Both the transformation for lexical attributes and the transformation for unstable subtypes from [13] can be obtained by composing one aggregation and one objectify operation.

Example 12. The following class hierarchy introduces a class Person, a class Employee, which inherits from class Person, and a class Company:

Class Person
Attributes
name : string
street : string
house : integer
city : string
Endclass
Class Employee **Isa** Person
Attributes
employer : Company
salary : integer
Endclass
Class Company
Attributes
name : string
Endclass.

The underlying type of class Person is:

$\mu\, t_P$. <id:oid, name:string, street:string, house:integer, city:string>.

The underlying type of class Person can be transformed into (using *aggregate* ({street, house, city}) (address)):

$\mu\, t_P$. <id:oid, name:string, address:<street:string, house:integer, city:string>>,

which can be transformed into (using *objectify*):

$\mu\, t_P$. <id:oid, name:string, address:$\mu\, t_X$. <id: oid, street:string, house:integer, city:string>>.

The composite transformation (F) is a variant of the transformation for lexical attributes from [13]. We can redefine class Person as a class (named Person1) that refers to a new class (named X).

Class Person1
Attributes
name : string
address : X
Endclass
Class X
Attributes
street : string
house : integer
city : string
Endclass.

Since the identities of the objects in class Person become the identities of the objects in the redefined class (Person1), the underlying constraint of class Person (*constr*) is preserved by the redefined class (w.r.t. F): the underlying constraint of the redefined class implies F(*constr*). The underlying type of class Employee is:

$\mu\, t_E$. <id:oid, name:string, street:string, house:integer, city:string,
employer:τ_C, salary:integer>,

where τ_C is the underlying type of class Company. The underlying type of class Employee can be transformed into (using *aggregate* ({id, name, street, house, city}) (employee)):

$\mu\, t_Y$. <employee:$\mu\, t_E$. <id:oid, name:string, street:string, house:integer, city:string>,
employer:τ_C, salary:integer>,

which can be transformed into (using *objectify*):

$\mu\, t_Y$. <id:oid, employee:$\mu\, t_E$. <id:oid, name:string, street:string, house:integer, city:string>,
employer:τ_C, salary:integer>.

The composite transformation F is a variant of the transformation for unstable subtypes from [13]. We can redefine class Employee as a 'relation' (named Y) that refers to a new class (named Employee1):

Class Y
Attributes
employee : Employee1
employer : Company
salary : integer
Endclass
Class Employee1
Attributes
name : string
street : string
house : integer
city : string
Endclass.

Since the identities of the objects in class Employee become the identities of the objects in class Employee1, and not the identities of the objects in the redefined class (Y), the underlying constraint of class Employee (*constr*) is not preserved by the redefined class (w.r.t. F): the underlying constraint of the redefined class and the rule 'x.employee.id = y.employee.id $\Rightarrow$ x.employee = y.employee' do not imply F(*constr*)). Therefore, we introduce a key for class Y:

Class Y1
Attributes

employee : Employee1
employer : Company
salary : integer
Constraints
key employee
Endclass.

The underlying constraint of class Y1 and the rule 'x.employee.id = y.employee.id $\Rightarrow$ x.employee = y.employee' do imply $F(constr)$). □

4 Application of schema transformations

In the previous section, we defined type transformations and showed how they induce schema transformations. In this section, we show how behaviour of methods can be used to choose among a set of schema transformations.
First, we define the set of factors of a class. A factor of a class C is a class that contains a part of the attributes, keys, and methods of C.

Definition 17 (Factors). Let H be a well-defined class hierarchy and C be a class in H. The set of factors of class C is defined as:

$$factors(C) = \{(d, \emptyset, A, K, M) \mid d \in CN \wedge A \subseteq atts(C) \wedge K \subseteq keys(C) \wedge M \subseteq meths(C) \wedge H \cup \{(d, \emptyset, A, K, M)\} \text{ is well-defined}\}.$$

□

Second, we give an example to illustrate that a class can be transformed in several ways, using different factors and different transformations.

Example 13. Let class Employee be the following class:

Class Employee
Attributes
name : string
dob : Date
street : string
house : integer
city : string
employer : Company
Methods
move (s:string,h:integer,c:string) =
street := s; house := h; city := c
Endclass

and class Address be a factor of Employee:

Class Address
Attributes
street : string
house : integer
city : string
Methods
move (s:string,h:integer,c:string) =
street := s; house := h; city := c
Endclass.

One option to transform class Employee is to redefine Employee as a subclass of Address (factorisation by specialisation):

Class Employee1 **Isa** Address
Attributes
name : string
dob : Date
employer: Company
Endclass.

Another option is to redefine Employee as a class referring to Address (factorisation by delegation):

Class Employee2
Attributes
name : string
dob : Date
address: Address2
employer: Company
Methods
move (s:string,h:integer,c:string) =
address := address.new_address(s,h,c)
Endclass
Class Address2
Attributes
street : string
house : integer
city : string
Methods
new_address (s:string,h:integer,c:string → l:Address2) =
l := **new**(Address2, street=s, house=h, city=c)
Endclass.

Note that, as an employee is not an address in the real world, it is unlikely that the first option is the right choice. The second option, where employee refers to an address (as one of its attributes) is a more reasonable choice.

Now, let class Person be a factor of class Employee2:

Class Person
Attributes
name : string
dob : Date
address : Address2
Methods
move (s:string,h:integer,c:string) =
address := address.new_address(s,h,c)
Endclass.

One option to transform class Employee2 is to redefine Employee2 as a subclass of Person (factorisation by specialisation):

Class Employee3 **Isa** Person
Attributes
employer : Company
Endclass.

Another option is to redefine Employee2 as a class referring to Person (factorisation by delegation):

Class Employee4
Attributes
person : Person1

employer : Company
Methods
move (s:string,h:integer,c:string) =
person := person.new_person(s,h,c)
Endclass
Class Person1
Attributes
name : string
dob : Date
address : Address2
Methods
new_person (s:string,h:integer,c:string → l:Person1)
l := **new**(Person1, name=name, dob=dob, address=address.new_address(s,h,c))
Endclass.

Since the identities of the objects in class Employee2 become the identities of the objects in class Employee4, we redefine method 'move' to be applicable to objects in class Employee4.

Yet another option is to redefine class Employee2 as a relation involving class Person:

Class Employment
Attributes
employee : Person
employer : Company
Constraints
key employee
Endclass.

Since the identities of the objects in class Employee2 become the identities of the objects in class Person, we do not redefine method 'move', because it is already applicable to objects in class Person.
Note that, as an employee is a person in the real world, it is likely that options one and three are more reasonable than option two, where an employer refers to a person (as one of its attributes).
□

As we have seen, a class can be transformed in several ways, using different factors and different transformations, e.g., factorisation by specialisation, factorisation by delegation, or redefinition as a relation. But how do we choose factors and how do we choose between specialisation, delegation and redefinition as a relation? For that purpose, we introduce evidence ratios for relatedness. Weak relatedness for a set of attributes says whether the attributes are mutually related (according to the methods). Strong relatedness for a set of attributes says whether the attributes are mutually related, but not to attributes outside the set (according to the methods). Isolation for a set of attributes says whether the attributes are not related to attributes outside the set (according to the methods).

Definition 18 (Relatedness ratios). Let H be a well-defined class hierarchy, C be a class in H, c be $name(C)$, and M be $meths(C)$. Furthermore, for $meth \in M$, let $atts(meth)$ consist of the names of attributes of C that occur in $meth$. Weak relatedness of a set of attributes $A \subseteq \{a \mid a : T \in atts(C)\}$ is defined as:

$$weakrel(c, A) = \frac{|\ \{meth \in M \mid atts(meth) \supseteq A\}\ |}{|\ \{meth \in M \mid atts(meth) \cap A \neq \emptyset\}\ |}.$$

Strong relatedness of a set of attributes A is defined as:

$$strongrel(c, A) = \frac{|\ \{meth \in M \mid atts(meth) = A\}\ |}{|\ \{meth \in M \mid atts(meth) \cap A \neq \emptyset\}\ |}.$$

Isolation of a set of attributes $A \subseteq \{a \mid a : T \in atts(C)\}$ is defined as:

$$isolation(c, A) = \frac{\mid \{meth \in M \mid \emptyset \neq atts(meth) \subseteq A\} \mid}{\mid \{meth \in M \mid atts(meth) \cap A \neq \emptyset\} \mid}.$$

If $\{meth \in M \mid atts(meth) \cap A \neq \emptyset\}$ is empty, then $weakrel(c, A)$ and $strongrel(c, A)$ are defined to be 0, and $isolation(c, A)$ is defined to be 1.

For a set of attributes with strong relatedness ratio 1 and any method, either all attributes occur in the method and all attributes that occur in the method are in the set, or no attribute in the set occurs in the method. In that case, the attributes are strongly related. For a set of attributes with weak relatedness ratio 0, there is no method in which all attributes occur and, hence, the attributes are not (mutually) related. And for a set of attributes with isolation ratio 1 and any method, either all attributes that occur in the method are attributes in the set or no attribute that occurs in the method is an attribute in the set. In that case, the attributes are only related within the set. □

Weak and strong relatedness can help to choose a factor. If the strong relatedness ratio of a set of attributes is high, then it is reasonable to believe that they belong together and, hence, to factorise. On the other hand, if the weak relatedness ratio is low, then it reasonable to believe that they do not belong together and, hence, not to factorise.

Example 14. Consider class Employee of Example 13. The weak and strong relatedness ratios for street, house, city and name, dob are given by:

$strongrel(\text{Employee}, \{\text{street}, \text{house}, \text{city}\}) = 1$
$weakrel(\text{Employee}, \{\text{street}, \text{house}, \text{city}\}) = 1$
$strongrel(\text{Employee}, \{\text{name}, \text{dob}\}) = 0$
$weakrel(\text{Employee}, \{\text{name}, \text{dob}\}) = 0.$

As we can see, street, house, and city are strongly related, whereas name and dob are not related.

Now, consider class Employee2 of Example 13. The weak and strong relatedness ratios for name, dob, address and name, dob, employer are given by:

$strongrel(\text{Employee2}, \{\text{name}, \text{dob}, \text{address}\}) = 0$
$weakrel(\text{Employee2}, \{\text{name}, \text{dob}, \text{address}\}) = 0$
$strongrel(\text{Employee2}, \{\text{name}, \text{dob}, \text{employer}\}) = 0$
$weakrel(\text{Employee2}, \{\text{name}, \text{dob}, \text{employer}\}) = 0.$

As we can see, in both cases the attributes are not related. □

Isolation can help to choose between specialisation and redefinition as a relation. If the isolation ratio is less than one, then specialisation is possible, but redefinition as a relation is not, since, in that case, we have to add a method to the relation that updates another relation or class.

Example 15. Consider class Employee2 of Example 13. The isolation evidence ratio for name, dob, address is given by:

$isolation(\text{Employee2}, \{\text{name}, \text{dob}, \text{address}\}) = 1.$

Redefinition as a relation results in a relation (Employment) that represents a simple association between a person and a company. Now, if we add a method to class Employee2 that updates attribute address and attribute employer, then we will have to add a method to Employment that creates a new person and updates attribute employee and attribute employer. However, this method inserts objects into a class different from the relation and should therefore not be associated with the relation. □

So, how do we choose factors and transformations? Factors are chosen by comparing weak evidence ratios. If the weak evidence ratio of a set of attributes is greater than some threshold, there is reason to assume that the attributes can be used as a factor. If not, there is no reason. Transformations are chosen by comparing strong evidence ratios and isolation ratios. In case the strong evidence ratio is greater than some threshold, delegation is a reasonable option, because the attributes are strongly related within the set and weakly related with other attributes. In case the isolation ratio is less than one, then specialisation is possible, but redefinition as a relation is not. Otherwise, specialisation or redefinition as a relation are both possible. It should be mentioned that, in the context of schema integration, schema transformations must be applied carefully and only if necessary. In particular, this is true for factorisation by specialisation, since a lot of new classes will be generated by this type of transformation.

The considerations for choosing factors and transformations can be used in a heuristic algorithm to support schema integration. First, the attributes of every class are partitioned in such a way that the isolation ratio of every element in the partition is one, and every class is factorised by delegation if desirable. Subsequently, for every pair of promising classes, a set of possible superclasses is computed, and both classes are factorised by specialisation or redefined as a relation if desirable.

Algorithm 1. The following algorithm is a heuristic for integrating two database schemas (resp., DBS1 and DBS2), given thresholds for strong relatedness and weak relatedness (resp., TSR and TWR):

```
integrate(DBS1,DBS2,TSR,TWR) =
    for every class C in DBS1∪DBS2
    do  for every element A in partition(C)
        do  if strongrel(c,A) ≥ TSR and 1 < |A| < |atts(C)|
            then create class C1 as the class containing A and the methods that refer to A;
                 factorise C by delegation using C1;
                 mark C and C1
            elif weakrel(c,A) ≥ TWR
            then mark C
            fi
        od
    od;
    for every marked C1 in DBS1
    do  for every marked C2 in DBS2
        do  if there is a superclass C of a class in joins(C1,C2) that can be used
            as a factor according to the designer
            then transform(C1,C2,C)
            else for every key a.p in keys(C1) such that a:D in atts(C1) for some class D
                 do define class D1 as obtained from C1 by applying
                        the inverse of redefinition as a relation;
                        if there is a superclass C of a class in joins(D1,C2) that can be used
                        as a factor according to the designer
                        then transform(D1,C2,C)
                        fi
                 od
            fi
        od
    od;

transform(C1,C2,C) =
    begin let φ1 be an injection from atts(C) to atts(C1) induced by C1 ⪯ C;
          let φ2 be an injection from atts(C) to atts(C2) induced by C2 ⪯ C;
          define A1 as the attribute names in the range of φ1;
```

define A2 as the attribute names in the range of φ_1;
if isolation(name(C1),A1) $<$ 1 or isolation(name(C2),A2) $<$ 1
then factorise C1 and C2 by specialisation using C
elif 1 $<$ |A1| $<$ |atts(C1)| and 1 $<$ |A2| $<$ |atts(C2)|
then factorise C1 and C2 or redefine C1 and C2 as relations
according to the choice of the designer
else factorise C1 and C2 by specialisation using C
fi
end;

where partition(C) is constructed as follows:

graph(C) has a node for every attribute name in atts(C)
graph(C) has an edge between two nodes if there is a method in meths(C)
in which both attribute names occur
partition(C) consists of sets of attribute names,
one set for every connected subgraph of graph(C):
two attribute names are in the same set if their nodes are connected
two attribute names are in different sets if their nodes are not connected,

and joins(D1,D2) (i.e., the set of common superclasses of D1 and D2) and $\preceq$ (i.e., the subclass relation) are as defined in [25, 24]. □

Note that the algorithm interacts with the designer. It should be mentioned again that the algorithm is a heuristic and should therefore be used in close interaction with the designer. The heuristic can be improved by combining the different thresholds and refining the different actions. This is the subject of future research. We conclude this section with an example.

Example 16. The following class hierarchy is a part of a drawing tool:

Class Square
Attributes
pos:Pos
width:int
Methods
set (x:int, y:int) =
pos := pos.set_pos(x,y)
translate (dx:int, dy:int) =
pos := pos.translate_pos(dx,dy)
Endclass
Class Pos
Attributes
x_co:int
y_co:int
Methods
set_pos (x:int, y:int → p:Pos) =
p := **new**(Pos, x_co=x, y_co=y)
translate_pos (dx:int, dy:int → p:Pos) =
p := **new**(Pos, x_co=x_co+dx, y_co=y_co+dy)
Endclass.

Objects in class Square have a position on the screen and a width, and can be moved around on the screen using method set and method translate.
The following class hierarchy is a part of another drawing tool:

Class Rectangle
Attributes

pos_x:int
pos_y:int
width_x:int
width_y:int

Methods

put (x:int, y:int) =
pos_x := x; pos_y := y
move (delta_x:int, delta_y:int) =
pos_x := pos_x + delta_x; pos_y := pos_y + delta_y

Endclass.

Objects in class Rectangle have a position w.r.t the x-axis, a position w.r.t. the y-axis, a length, and a width, and can be moved around on the screen using method put and method move.
Let C_S be class Square, C_P be class Pos, and C_R be class Rectangle. The partitions of these classes are given by:

$\text{partition}(C_S) = \{\{\text{pos}\}, \{\text{width}\}\}$
$\text{partition}(C_P) = \{\{\text{x\_co}, \text{y\_co}\}\}$
$\text{partition}(C_R) = \{\{\text{pos\_x}, \text{pos\_y}\}, \{\text{width\_x}\}, \{\text{width\_y}\}\}$.

Now, let us use Algorithm 1 to integrate the class hierarchies. In the first big loop, all classes are marked and Rectangle is factorised using delegation:

Class Rectangle
Attributes
pos:X
width_x:int
width_y:int
Methods
put (x:int, y:int) =
pos := pos.put_pos (x,y)
move (dx:int, dy:int) =
pos := pos.move_pos (dx,dy)
Endclass
Class X
Attributes
pos_x:int
pos_y:int
Methods
put_pos (x:int, y:int → l:X) =
l := **new**(X, pos_x=x, pos_y=y)
move_pos (dx:int, dy:int → l:X) =
l := **new**(X, pos_x=pos_x+dx, pos_y=pos_y+dy)
Endclass.

Let C'_R be the new class Rectangle and C_X be class X. In the second big loop, the only interesting join sets are:

$\text{joins}(C_P, C_X) = \{\gamma_P(C_P)\} = \{\gamma_X(C_X)\}$
$\text{joins}(C_S, C'_R) = \{\gamma_S(C_S)\}$,

where the γ's are rename functions replacing attribute names and method names.

Suppose the designer chooses C_P as a factor to factorise C_P and C_X (redefinition as a relation is not possible). Since C_P and C_X are equivalent, it suffices to remove C_X and redefine every class D that refers to C_X by replacing the occurrences of X in D by Pos and redefine every method in D in which an attribute or method name *name* occurs that refers to an attribute or method

of C_X by replacing these occurrences of *name* by the corresponding attribute or method name in Pos. Let C''_R be the redefined class Rectangle and suppose the designer chooses the following class as a factor to factorise C_S and C''_R (again, redefinition as a relation is not possible):

Class Figure
Attributes
    pos:Pos
Methods
    set (x:int, y:int) =
        pos := pos.set_pos (x,y)
    translate (dx:int, dy:int) =
        pos := pos.translate_pos (dx,dy)
Endclass.

Then the integrated class hierarchy will consist of class Pos, class Figure, and:

Class Square **Isa** Figure
Attributes
    width:int
Endclass
Class Rectangle **Isa** Figure
Attributes
    width_x:int
    width_y:int
Endclass.

Note that the only real choice made by the designer is the choice to factorise Square and Rectangle using Figure instead of Square. □

5 Conclusion

In this report, we presented a new approach to schema integration based on transformations and behaviour. First, we formalised schemas using underlying types and underlying constraints. Next, we presented a number of type transformations on underlying types and used them to transform schemas. Finally, we gave a heuristic algorithm for integrating schemas. The algorithm uses schema transformations to restructure schemas and join operators to merge them and behavioural information to guide restructuring and merging. Advantages of this approach are: structural aspects are integrated in a guided fashion, and both structural and behavioural aspects are integrated.

Further research includes extension of the data model, extension of the set of schema transformations, properties of the schema transformations, and extension and refinement of the heuristic algorithm.

A Meet types

Let H be a class hierarchy. Furthermore, let T and T' be types that occur in H. The meet of T and T', denoted by $T \sqcap T'$, is defined as follows (cf. [8]):

1. $T, T' \in CN, T = name(D_1), T' = name(D_2)$:
   (a) $T \sqcap T' = T$ if $T = T'$,
   (b) $T \sqcap T' = d$ is a new class name corresponding to class:

   $(d, \emptyset, \{a_1 : T_1 \in atts(D_1) \mid \forall a_2 : T_2 \in atts(D_2)[a_1 \neq a_2]\}$
   $\cup \{a_2 : T_2 \in atts(D_2) \mid \forall a_1 : T_1 \in atts(D_1)[a_1 \neq a_2]\}$
   $\cup \{b : U \mid \exists a_1 : T_1 \in atts(D_1) \exists a_2 : T_2 \in atts(D_2)[b = a_1 = a_2 \wedge U = T_1 \sqcap T_2\}, \emptyset)$,
   where $T_1 \sqcap T_2 = d$ if
   $T_1 = name(D_1) \wedge T_2 = name(D_2)$ or $T_1 = name(D_2) \wedge T_2 = name(D_1)$,

   if $\forall a_1 : T_1 \in atts(D_1) \forall a_2 : T_2 \in atts(D_2)[a_1 = a_2 \Rightarrow T_1 \sqcap T_2 \neq \bot]$
   (c) $T_1 \sqcap T_2 = \bot$ otherwise
2. $T, T' \in \{\text{integer}, \text{rational}, \text{string}\}$: $T \sqcap T' = T$ if $T = T'$ and $T \sqcap T' = \bot$ otherwise
3. $T = \{U\}, T' = \{U'\}$: $T \sqcap T' = \{U \sqcap U'\}$ if $U \sqcap U' \neq \bot$ and $T \sqcap T' = \bot$ otherwise
4. $T = < l_i : U_i \mid i \in I >, T' = < l_i : U'_i \mid i \in I' >$:
   (a) $T \sqcap T' =$
   $< l_i : U \mid (i \in I - I' \wedge U = U_i) \vee (i \in I' - I \wedge U = U'_i) \vee (i \in I \cap I \wedge U = U_i \sqcap U'_i) >$
   if $\forall i \in I \cap I'[U_i \sqcap U'_i \neq \bot]$
   (b) $T \sqcap T' = \bot$ otherwise
5. otherwise: $T \sqcup T' = \bot$.

Since $\sqcap$ is commutative and associative (modulo class renaming), $\sqcap\{T_1, \cdots, T_n\} = T_1 \sqcap \cdots \sqcap T_n$ is well-defined.

B Referential integrity formulas

Let H be a class hierarchy, C be a class in H, c be $name(C)$, and $a : T$ be an attribute in $atts(C)$ that refers to C. Furthermore, let T^* be a type obtained from T by labelling one occurrence of c with $*$. The access path to $c*$ in T^*, denoted by $path(c*, T^*)$, is defined as follows:

$path(c*, c*) = empty$
$path(c*, \{c*\}) = \in$
$path(c*, < \cdots, l : c*, \cdots >) = l$
$path(c*, \{U^*\}) = \in .path(U^*)$
$path(c*, < \cdots, l : U^*, \cdots >) = l.path(U^*)$,

where $c*$ occurs in U^*. Then every occurrence of c in $a : T$ is uniquely defined by its access path. Finally, the referential integrity formula corresponding to an occurrence of c in $a : T$ with access path p is defined as:

$\forall x \in e\ [refint(x.a, p, 0)]$

where

$refint(p, empty, i) = \exists y \in e\ [p = y]$
$refint(p, l, i) = \exists y \in e\ [p.l = y]$
$refint(p, \in, i) = \forall x_{i+1} \in p\ \exists y \in e\ [x_{i+1} = y]$
$refint(p, l.q, i) = refint(p.l, q, i)$
$refint(p, \in .q, i) = \forall x_{i+1} \in p\ refint(x_{i+1}, q, i + 1)$.

C Types of sources and destinations

Let H be a class hierarchy and C be a class in H. Furthermore, let $m(P \to l : T) = E$ be a query method in $meths(C)$. We define the type of a source or destination of an assignment in E as follows:

- the type of **self** is $name(C) = c$
- if $s : B$ is a parameter in P, then the type of s is B
- if $s = l.r_1.\cdots.r_n$ is the labeling of a path in $struc(< l : type(T) >)$, starting at the root, then the type of s is $T.r_1.\cdots.r_n$
- if $s = a_i.r_1.\cdots.r_n$ is the labeling of a path in $struc(type(C))$, starting at the root, then the type of s is $T_i.r_1.\cdots.r_n$
- the type of a constant $v \in ext(B)$ is B
- if s is an expression, then the type of s follows from the types of the subexpressions and the standard rules for $+$ (addition for integers and rationals; concatenation for strings), $-$ (subtraction), $\times$ (multiplication), and $\div$ (division)
- the type of $\mathbf{new}(d, e_1, \cdots, e_n)$ is d
- otherwise, the type of a source or destination is undefined

where

$< \cdots, l = T, \cdots > .l = T$
$c'.a = T$ if $\exists C' \in H\ [name(C') = c' \wedge a : T \in atts(C')]$.

Now, let $m(P) = E$ be an update method in $meths(C)$. The type of a source or destination of an assignment in E is defined as above, with the exception that the third item (regarding selections of the result of a query method) is removed and the following item (regarding method calls) is added:

the type of a call to method $m'(P' \to l' : T') = E'$ is T'.

D Syntactic evaluation of method bodies

Let H be a class hierarchy, C be a class in H, c be $name(C)$, and $\{a_1 : T_k, \cdots, a_k : T_k\}$ be $atts(C)$. Furthermore, let $m(P \to l : T) = E$ be a query method in $meths(C)$. The syntactic evaluation of body E in state $\sigma = (obj, r, f)$ (where obj is the object that is queried by the method, r is the current value of the result of the method, and f is the function that, for every class C in H, gives the current number of objects in class C), denoted by $eval(E)\sigma$, is given by:

$eval_q(L_1; L_2)\sigma = eval_q(L_2)(eval_q(L_1)\sigma)$,
$eval_q(l := s)(obj, r, f) = (obj, \pi_1(ev(s)(obj, r, f)), \pi_2(ev(s)(obj, r, f)))$,
$eval_q(l.l_1.\cdots.l_n := s)(obj, r, f) = (obj, r[l_1.\cdots.l_n = \pi_1(ev(s)(obj, r, f))], \pi_2(ev(s)(obj, r, f)))$,
$eval_q(\mathbf{insert}(s, l))(obj, r, f) = (obj, r \cup \{\pi_1(ev(s)(obj, r, f))\}, \pi_2(ev(s)(obj, r, f)))$,
$eval_q(\mathbf{insert}(s, l.l_1.\cdots.l_n))(obj, r, f) = (obj, r[l_1.\cdots.l_n = (r.l_1.\cdots.l_n \cup \{\pi_1(ev(s)(obj, r, f))\})]$,
$\quad \pi_2(ev(s)(obj, r, f)))$,

where

$ev(\mathbf{self})(obj, r, f) = (obj, f)$,
$ev(a.l_1.\cdots.l_n)(obj, r, f) = (obj.a.l_1.\cdots.l_n, f)$
$\quad$ if $a \in AN$,
$ev(l_1.\cdots.l_n)(obj, r, f) = (l_1.\cdots.l_n, f)$

if $l_1 \in L$,
$ev(b)(obj, r, f) = (b, f)$,
if $b \in Cons$
$ev(s_1 \theta s_2)(obj, r, f) = (\pi_1(ev(s_1)(obj, r, f))\theta\pi_1(ev(s_2)(obj, r, f)), f)$
if $\theta \in \{+, -, \times, \div\}$
$ev(\mathbf{new}(c, a_1 = e'_1, \cdots, a_k = e'_k))(obj, r, f) =$
$(\mu y. <\mathrm{id} = newid(c, f(c)), a_1 = \pi_1(ev(e'_1)(obj, r, f)), \cdots a_k = \pi_1(ev(e'_k)(obj, r, f)) >, f')$,
where $newid : CN(H) \times \mathrm{integer} \rightarrow \mathrm{oid}$ is some injective function
and f' is equal to f, except that $f'(c) = f(c) + 1$
and $ev(\mathbf{nself})(obj, r, f)$ is (y, f)

and

$$< l_1 = v_1, \cdots, l_n = v_n > [l_1 = v] = < l_1 = v, \cdots, l_n = v_n >$$
$$< l_1 = v_1, \cdots, l_n = v_n > [l_1.l'_1. \cdots .l'_n = v] = < l_1 = v_1[l'_1. \cdots .l'_n = v], \cdots, l_n = v_n >.$$

Now, let $m(P) = E$ be an update method in $meths(C)$. The syntactic evaluation of body E in state $\sigma = (\mu x.e, f)$ (where $\mu x.e$ is the object that is updated by the method and f is the function that, for every class C in H, gives the current number of objects in class C), denoted by $eval(E)\sigma$, is given by:

$eval_u(L_1; L_2)\sigma = eval_u(L_2)(eval_u(L_1)\sigma)$,
$eval_u(a_1 := s)(\mu x.e, f) =$
$(\mu x.(reduce_x(<\mathrm{id} = e_0, a_1 = \pi_1(ev(s)(\mu x.e, f)), \cdots, a_k = e_k >)), \pi_2(ev(s)(\mu x.e, f)))$,
$eval_u(a_1.l_1. \cdots .l_n := s)(\mu x.e, f) =$
$(\mu x.(reduce_x(<\mathrm{id} = e_0, a_1 = e.a_1[l_1. \cdots .l_n = \pi_1(ev(s)(\mu x.e, f))], \cdots, a_k = e_k >)),$
$\pi_2(ev(s)(\mu x.e, f)))$,
$eval_u(\mathbf{insert}(s, a_1))(\mu x.e, f) =$
$(\mu x.(reduce_x(<\mathrm{id} = e_0, a_1 = e.a_1 \cup \{\pi_1(ev(s)(\mu x.e, f))\}, \cdots, a_k = e_k >)),$
$\pi_2(ev(s)(\mu x.e, f)))$,
$eval_u(\mathbf{insert}(s, a_1.l_1. \cdots .l_n))(\mu x.e, f) =$
$(\mu x.(reduce_x(<\mathrm{id} = e_0, a_1 = e.a_1[l_1. \cdots .l_n = e.a_1.l_1. \cdots .l_n \cup \{\pi_1(ev(s)(\mu x.e, f))\}],$
$\cdots, a_k = e_k >)), \pi_2(ev(s)(\mu x.e, f)))$,

where

$ev(\mathbf{self})(\mu x.e, f) = (x, f)$,
$ev(a.l_1. \cdots .l_n)(\mu x.e, r, f) = (e.a.l_1. \cdots .l_n, f)$
if $a \in AN$,
$ev(l_1. \cdots .l_n)(\mu x.e, r, f) = (l_1. \cdots .l_n, f)$
if $l_1 \in L$,
$ev(b)(\mu x.e, r, f) = (b, f)$
if $b \in Cons$
$ev(s_1 \theta s_2)(\mu x.e, r, f) = (\pi_1(ev(s_1)(\mu x.e, r, f))\theta\pi_1(ev(s_2)(\mu x.e, r, f)), f)$
if $\theta \in \{+, -, \times, \div\}$,
$ev(s.m'(e'_1, \cdots, e'_n))(\mu x.e, f) =$
$\pi_{2,3}([|\ m'\ |](f)(\pi_1(ev(s)(\mu x.e, f)))(\pi_1(ev(e'_1)(\mu x.e, f)))(\pi_1(ev(e'_n)(\mu x.e, f))))$,
where $[|\ m'\ |]$ is the functional form of method m' in the class to which s refers

and

$reduce_x(e') = < l_1 = reduce_x(e'_1), \cdots, l_n = reduce_x(e'_n) >$
if $e' = < l_1 = e'_1, \cdots, l_n = e'_n >$ for some $e'_1, \cdots, e'_n$,
$reduce_x(e') = x$ if $e' = \mu x.e'_1$ for some e'_1,
$reduce_x(e') = e'$ otherwise.

E Bibliography

[1] S. Abiteboul and R. Hull. Restructuring hierarchical database objects. *Theoretical Computer Science*, 62:3–38, 1988.

[2] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly typed, interactive conceptual language. *ACM Trans. on Database Systems*, 10(2):230–260, 1985.

[3] R. Amadio and L. Cardelli. Subtyping recursive types. In *Proc. Int. Symp. on Principles of Programming Languages*, pages 104–118, 1991.

[4] P. Apers, H. Balsters, R. de By, and C. de Vreeze. Inheritance in an object-oriented data model. Memoranda Informatica 90-77, University of Twente, Enschede, The Netherlands, 1990.

[5] H. Balsters, R. de By, and R. Zicari. Typed sets as a basis for object-oriented database schemas. In *Proc. Computing Science in the Netherlands*, pages 62–77. Stichting Mathematisch Centrum, Amsterdam, The Netherlands, 1991.

[6] C. Batini and M. Lenzerini. A methodology for data schema integration in the ER model. *IEEE Transactions on Software Engineering*, pages 650–664, November 1984.

[7] C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[8] L. Cardelli. A semantics of multiple inheritance. In *Proc. Int. Symp. on Semantics of Datatypes, LNCS 173*, pages 51–67. Springer-Verlag, Berlin, 1984.

[9] E. Casais. An incremental class reorganization approach. In *European Conf. on Object-Oriented Programming*, pages 114–132, 1992.

[10] C. de Vreeze. Formalization of inheritance of methods in an object-oriented data model. Memoranda Informatica 90-76, University of Twente, Enschede, The Netherlands, 1990.

[11] R. Elmasri and G. Wiederhold. Data model integration using the structural model. In *Proc. Int. Conf. on Management of Data*, pages 191–202, 1979.

[12] P. Fankhauser, M. Kracker, and E. Neuhold. Semantic vs. structural resemblance of classes. *ACM SIGMOD Record*, 20(4):59–63, 1991.

[13] P. Johannesson. Schema transformations as an aid in view integration. In *Proc. Int. Conf. on Advanced Information Systems Engineering, LNCS 685*, pages 71–92. Springer-Verlag, Berlin, 1993.

[14] M. Kersten. Goblin: a DBPL designed for advanced database applications. In *Proc. Int. Conf. on Database and Expert Systems Applications*, pages 345–349. Springer-Verlag, Wien, 1991.

[15] C. Koster. On infinite modes. *ACM SIGPLAN Notices*, 4(3):109–112, 1969.

[16] J. Larson, S. Navathe, and R. Elmasri. A theory of attribute equivalence in databases with application to schema integration. *IEEE Transactions on Software Engineering*, 15(4):449–463, 1989.

[17] C. Lécluse and P. Richard. The O_2 database programming language. In *Proc. Int. Conf. on Very Large Databases*, pages 411–422. Morgan Kaufmann, Palo Alto, CA, 1989.

[18] M. Mannino, S. Navathe, and W. Effelsberg. A rule based approach for merging generalisation hierarchies. *Information Systems*, 13(3):257–272, 1988.

[19] A. Motro and P. Buneman. Constructing superviews. In *Proc. Int. Conf. on Management of Data*, pages 56–64, 1981.

[20] S. Navathe and S.Gadgil. A methodology for view integration in logical data base design. In *Proc. Int. Conf. on Very Large Databases*, pages 142–155, 1982.

[21] A. Sheth and S. Gala. Attribute relationships: an impediment in automating schema integration. In *Proc. Workshop on Heterogeneous Database Systems*, 1989.

[22] M. Siegel and S. Madnick. A metadata approach to resolving semantic conflicts. In *Proc. International Conference on Very Large Databases*, pages 133–145, 1991.

[23] C. Thieme and A. Siebes. Schema integration in object-oriented databases. In *Proc. Int. Conf. on Advanced Information Systems Engineering, LNCS 685*, pages 54–70. Springer-Verlag, Berlin, 1993.

[24] C. Thieme and A. Siebes. Schema refinement and schema integration in object-oriented databases. In *Proc. Computing Science in The Netherlands*, pages 343–354. Stichting Mathematisch Centrum, 1993.

[25] C. Thieme and A. Siebes. Schema refinement and schema integration in object-oriented databases. Report CS-R9354, CWI, Amsterdam, The Netherlands, 1993.

[26] C. Yu, W. Sun, S. Dao, and D. Keirsey. Determining relationships among attributes for interoperability of multi-database systems. In *Proc. Int. Workshop on Interoperability in Multidatabase Systems*, pages 251–257, 1991.